Generating SGML specific editors: from DTDs to Attribute Grammars

José Carlos Ramalho Alda Reis Lopes Pedro Rangel Henriques

30 de Setembro de 2004

Resumo

SGML (Standard Generalized Markup Language) is well established in electronic publishing industry. The number of users and the number of applications grows everyday.

If we look at the market the choice of available tools is very wide. We have tools for every purpose and for each price.

However, from a technical point of view there are still some open areas for research and improvement. In this work we will address the parsing technology. Can we improve it? Or, is there another way to do it? Can we achieve better results? These are some of the questions that we will try to answer along this paper.

SGML parsers are normally implemented using traditional technology: syntax directed translation. In the SGML context these parsers are good and offer a good performance. The problems emerge when we want to do something else besides structural validation. For example, to build an extension to perform semantic checking or to have on-line validation instead of batch validation.

The focus of this paper will be a DTD (Document Type Definition) editor that generates specific editors for each specific type of document. To implement this editor we develop an AG (Attribute Grammar) for SGML syntax. But the most important part it is not this grammar. The editor has a built-in generator that at any moment translates the DTD being edited to a generic AG.

In this paper we will discuss the methodology used to develop this DTD editor and we will make a detailed presentation of the conversion performed between DTDs and Attribute Grammars. At the end we will show a glimpse of the intended environment that along with this editor includes a style editor and a semantics editor.

1 Introduction

The issues addressed in this paper are related to SGML parsing technology, the core of every comercial SGML editing environment.

The main idea is to explore new methodologies and compare the results to see if we can improve what we already have.

This does not mean that the actual technology has problems. It has served well but is gradually becoming insufficient.

Looking at the Computer Science universe we can find another field with an evolution very similar to SGML: Programming Languages. Going deeper in our analysis we can reach the conclusion that programs written in a specific programming language and SGML documents are almost equivalent:

- Programs have a support language defined by a formalism – SGML documents are written with a markup language defined also formally
- In the definition of a programming language we have the lexical definition (the vocabulary) and the syntactic definition expressed by a CFG (Context Free Grammar) – a markup language is specified by a SGML declaration and a DTD.

Besides that relationship between a CFG and a DTD, we can take this comparison a little further, to the processing of SGML documents and programs. SGML documents as programs have to be analysed for subsequent transformation. In order to do this a lexical analysis followed by a syntactic analysis have to be performed. This applies to both. The difference comes when we move towards the semantic analysis. Actual SGML parsers are only able to perform some simple validations like the one related to the ID and IDREF attributes.

Concerning SGML, we are at a stage already overcome in the programming languages universe. We want to have more control over document content; we need to be able to express constraints over element content and to process them. This will only be possible by changing the way we are processing SGML, using more powerful methodologies and technologies.

In the early 70's we were facing the same needs and the same difficulties with programs. The answer was to move towards a new model of language processing, *Semantics Directed Translation*. This model is based in attribute grammars; it uses attributes and attribute equations to express semantics and attribute evaluators to process them. This paradigm brought new concepts with it, such as incremental parsing and incremental editing with on-line validation.

In the work we are currently developing we are facing the need to process semantics. We also want to explore new ways and new ideas. So we decide to develop an environment based on Attribute Grammars to process SGML documents. In this environment we aim at building automatically specific editors for each document type (according to the respective DTD). Specific editors tend to be more efficient and easy to use then generic ones. Moreover the proposed specific syntax-directed editors are acopulated with an incremental compiler that enables to perform on-line syntatic and semantic validation.

In the next sections we explain the central piece of our system: a DTD structured editor that generates attribute grammars for specific editors. But first we introduce the attribute grammar formalism, and the tool used to implement attribute grammars, SGen (Synthesizer Generator; then we present the architecture of our system INES.

2 Attribute Grammars

An AG is a well accepted formalism used by the compilers' community to specify the syntax and semantics of languages.

Introduced by Knuth [Knu68a], the AG appeared as an extension to the classic CFG to allow the local definition (without the use of global variables) of the meaning of each symbol in a declarative style.

Terminal symbols have intrinsic attributes (that describe their lexical information) and Nonterminal symbols are associated with generic attributes; semantic information can be synthesized up the tree (from the bottom to the root), but can also be inherited down the tree (from the top to the leaves), enabling explicit references to contextual dependencies.

Let G be a CFG defined as a tuple $G =< T, N, S, P >$, where:

- T denotes the set of terminal symbols (the alphabet)
- N is the set of nonterminal symbols
- S is the start symbol, or grammar axiom: $S \in N$
- P is the set of productions, or derivation rules, each one of the form:

$$A \longrightarrow \delta, A \in N \land \delta \in (T \cup N)*$$

  that we will represent in the sequel as:

$$X_0 \longrightarrow X_1 X_2 ... X_n$$

For each abstract production $p \in P$ — a derivation rule without keywords in its `Right Hand Side (rhs)` of the production — there is an associated abstract tree whose root is X_0, the nonterminal in its `Left Hand Side (lhs)`, and its n descendents are the X_i, $i \geq 1 \wedge i \leq n$ symbols in the rhs.

Given a sentence of L_G, the language generated by grammar G, an AST (Abstract Syntax Tree), also called *Abstract Derivation Tree*, is a tree whose root is S, the grammar start symbol, and the frontier (the leafs) is composed by the terminal-classes that once concatenated from left to right form the given sentence. The AST is built pasting the abstract trees corresponding to each grammar rule $p \in P$ used to derive the sentence from the axiom S.

An Attribute Grammar `AG` is a tuple $AG =< G, A, R, C >$, where:

- G is a CFG as defined above
- A is the union of $A(X)$ for each $X \in (T \cup N)$, and denotes the set of all attributes (each one has a name and a type); the attributes of a terminal symbol are called *intrinsic* and their value do not need to be evaluated; for each nonterminal X, its set of attributes $A(X)$ is splitted into two disjoint subsets: the inherited attributes $AI(X)$, and the synthesized attributes $AS(X)$
- R is the union of R_p, the set of *attribute evaluation rules* for each production $p \in P$
- C is the union of C_p, the set of *contextual conditions* for each production $p \in P$

Let p be a CFG production ($p \in P$), R_p its set of attribute evaluation rules, and C_p its set of contextual conditions; in this context:

- $a(X_i)$, $i \geq 0$, represents the attribute a associated to the symbol X that occurs in position i of production p
- an attribute evaluation rule is an expression of the form $a(X_i) = fun(..., b(X_j), ...)$ with:
  - $a \in AS(X_0) \vee a \in AH(X_i), i \geq 0$;
  - $b \in AH(X_0) \vee b \in AS(X_j), j \geq 0$ and
  - fun any function of type V_a (the type of attribute a)
- a contextual condition is a predicate of the form: $pred(..., a(X_i), ...), i \geq 0$

Given a sentence of L_{AG} — the language generated by the attribute grammar AG — let AST be its *Abstract Syntax Tree*, as defined above. Each tree *node* is, originally, labeled by the corresponding grammar symbol and the identifier of the production that was applied to derive it.

Assume, now, that each tree node is enriched with the attributes (either inherited or synthesized) associated to that symbol.

A *Decorated Abstract Syntax Tree* (DAST) is the initial AST after the attribute evaluation process, with all the attribute occurrences in each node associated with an actual value (belonging to its proper domain). To be a DAST it is necessary that all the contextual conditions, related to the production labeling each node, are satisfied (evaluate to true – for the actual attribute occurrence values).

We exemplify this abstract definitions with a fragment of the AG behind our system. In this example, we will signal with error messages duplicated element declarations in a DTD. To do that we use two attributes, one inherited (travelling top to bottom through the derivation tree) is named `ElemTab`, the other, is synthesized (travelling bottom to top through the derivation tree) and is named `ElemNewTab`. Both are used to store generic identifiers corresponding to declared SGML elements. In a more informal way we can say that inherited attributes are input parameters and synthesized attributes are output parameters.

```
DTD --> Declarations
        Declarations.ElemTab = ()
```

```
Declarations --> Declaration Declarations
        Declaration.ElemTab = Declarations.ElemTab
        Declarations$2.ElemTab = Declaration.ElemNewTab
    |  &epsi;

Declaration --> ElemDecl
    ElemDecl.ElemTab = Declaration.ElemTab
    Declaration.ElemTab = ElemDecl.ElemNewTab
                  | AttDecl
                  | ...

ElemDecl --> gi min min Content
    if ( not exist( gi, ElemDecl.ElemTab )
        ElemDecl.ElemNewTab = insert( ElemDecl.ElemTab, gi )
    else  error("Duplicated Element!!!")
```

We show below the DAST corresponding to the example above.

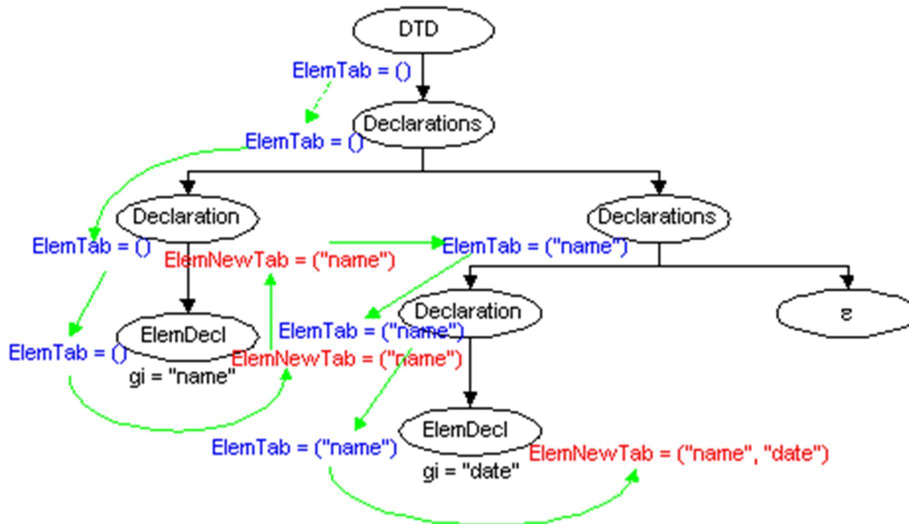

Figura 1: **Decorated Abstract Syntax Tree**

3 Synthesizer Generator

In the implementation of our environment we are using the *Synthesizer Generator (SGen)* [RT89a, RT89b].

SGen was initially developed to try out the incremental compiling paradigm based on Attribute Grammars. By *incremental compiling* we mean the ability to recompile a source text, after a change, with minium effort, without analysing the unchanged parts. Today, SGen is a tool widely used both in industry and academic environments.

SGEN is normally used as a structured editor generator. The generated editors can range between simple text editors to smart editors that have built-in inference mechanisms and complex output generators.

The authoring process in these editors is made in an incremental way, i.e., the user is immediately informed each time he gives an error being able to correct it in the moment.

SGen generates a structured editor and an incremental compiler for a language using an Attribute Grammar based specification. This specification is composed of:

- A lexical module, where the lexemes are defined.
- Two syntactic modules. SGen distinguishes between the abstract and concrete syntax. The abstract syntax is used to build the internal representation of the document being introduced. The concrete syntax is used to parse the document.
- One or more unparsing modules. Each unparsing module represents one view of the internal representation. This way, we will have as much of these modules as the number of layouts we want for each document being processed.
- A semantics module, where the attributes and their equations are specified.
- An optional tranformation module, where transformations are specified. These transformations will operate over the internal representation. The utility of these transformations is to provide shortcuts in the interface so the user can speed up the introduction of text without errors.

In the following sections we will follow the generation of the abstract grammar.

4 INES: a document programming environment

The work we are presenting is the first piece of a larger puzzle which we are building. The main idea was to explore the paralellism between SGML documents and programs and to create a document programming environment. We baptized the global system with the name `INES`.

In the next figure (fig. 2) we present the basic working architecture of INES.

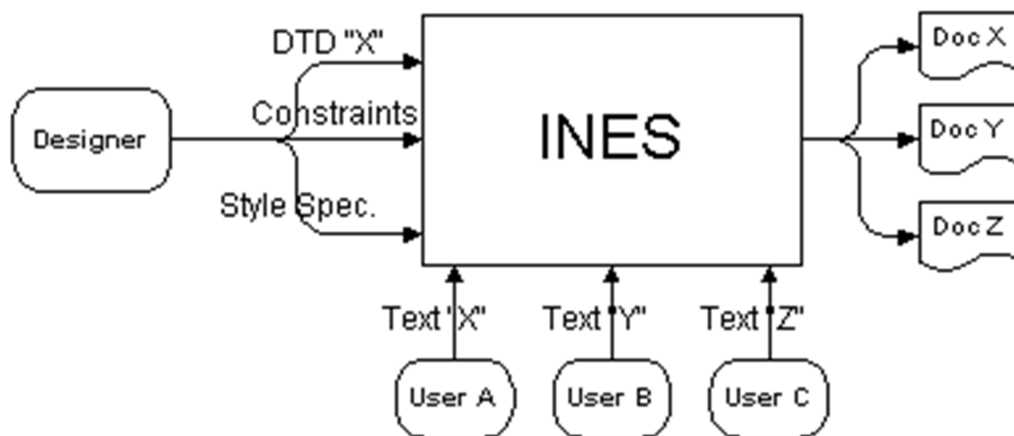

Figura 2: **INES: Document Programming Environment**

As one can see, we consider two kinds of people as the customers of INES: the document designer and the author. The designer will interact with INES to provide DTD definitions together with Style specifications (XSL — eXtensible Style Language) based language and Semantics specifications (in the form of constraints - specific language being developed) [RH98]. All these specifications are used to generate a specific editor (one specific editor is defined by one DTD, one style specification and an optional semantic specification). On the other hand, the author will interact with the generated editor to create his documents.

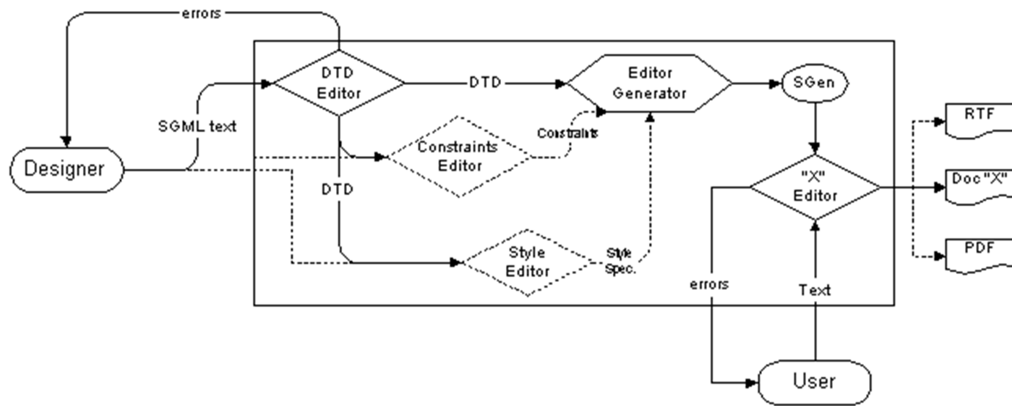

Figura 3: **INES: inside**

These components and their interactions are described in figure 3.

The elements represented in figure 3 by dotted lines are beyond the scope of this article and will soon be covered by other texts. In this article we are covering the syntactic part of the environment, represented in the diagram by the *DTD editor* and the *Editor Generator*.

This part of the system is composed of three modules:

- A DTD syntax-driven editor, generated by the SGen tool (given an AG for SGML);
- An incremental syntactic and semantic processor (compiler), integrated with the structured editor (not shown separately in the figure);
- A transformer, called "Editor Generator", that takes a DTD and converts it into an appropriate Attribute Grammar that will be sent to SGen to generate the new specific editor.

4.1 Working with the DTD editor

The DTD editor (a syntax-directed editor generated by SGen) provides the designer, at any time, with the set of SGML possible declarations (element, attribute, entity, ...). This set of SGML declarations is represented at the bottom of the editor window by a set of buttons as can be seen in figure 4.

When a declaration is selected, the editor immediately expands the declaration icon into the structure of that alternative, inserting all reserved words automatically. This way, the user only has to fill the variable parts of each declaration.

For example, if the user chooses the declaration `Element`, the editor will expand the symbol $< Declaration >$ into $<!Element < Name >< Min >< Min >< ElementContent >>$

Now the symbols `Name, Min and Element_content` can be selected and expanded the same way, until the derivation process reaches the bottom (a leaf of the derivation tree). For instance, if the user selects the symbol `Name` the editor will wait for the introduction of an identifier. This working scheme avoids lexical and syntactic errors that an absent-minded user could enter.

In addition to the lexical and syntactic validation, semantic checks can be made; all detected errors are reported in another window.

Figure 5 shows a complete DTD for a document of type `Letter`, created with the DTD editor described.

According to the internal representation of the DTD, the editor generator builds an Attribute Grammar based specification and sends it to SGen that will use it to generate the specific editor. With this new specific editor, the

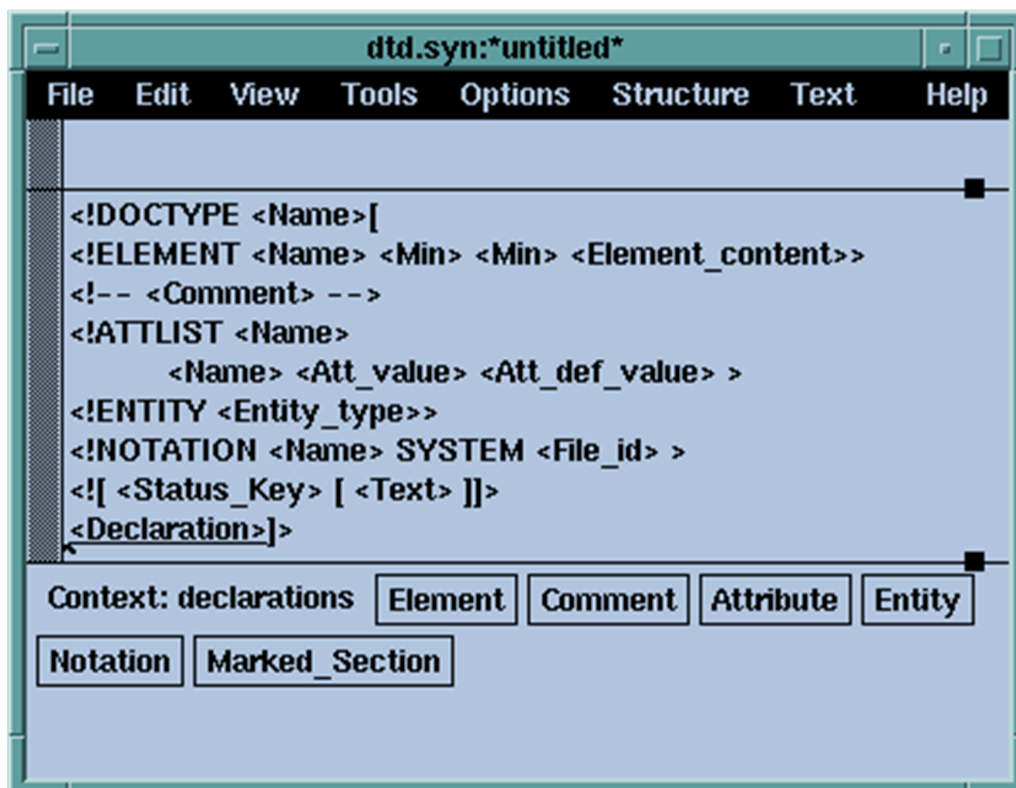

Figura 4: **DTD editor**

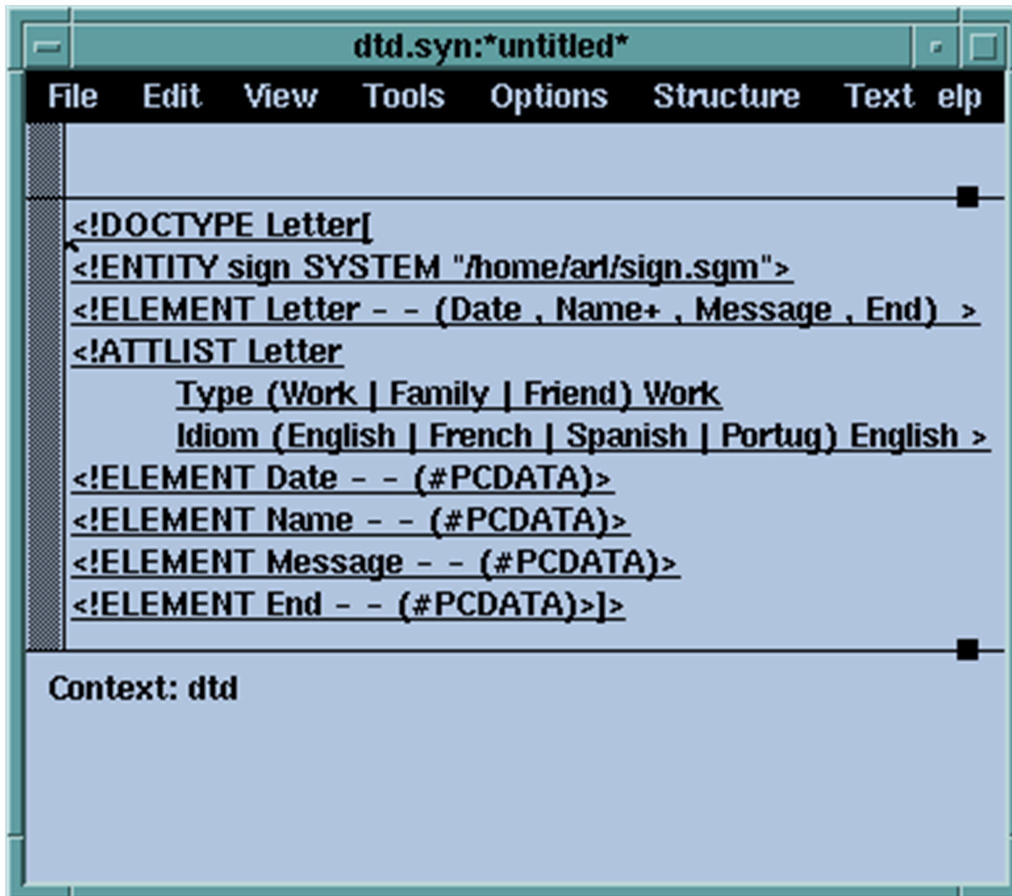

Figura 5: **Example: DTD for type *Letter***

user will be able to author documents of `Letter` type. The SGML documents of that type will have the structure illustrated by the following example:

```
<Letter Type="Friend" Idiom="English">
  <Date>30 July 1998</Date>
  <Name>Alexandra</Name>
  <Message>I wish you a good weekend!</Message>
  <End>Cheers!</End>
</Letter>
```

In the next section we will take a look at the process of attribute grammars generation.

5 Generating Attribute Grammars

To generate a new syntax-directed editor, SGen requires a description of the language to be edited in the form of a AG specification.

As described above, this AG specification should be composed of the standard SGen modules: the abstract grammar, the concrete grammar, the unparsing rules, the attribute definitions and equations, and the transformation rules.

The generation process is centered on the abstract grammar.

We shall now exemplify the generation process in two steps. In the first one we will issue a conversion of the SGML file presented above into the correspondent abstract grammar. Then we generalize this process and we show how to convert the DTD into an abstract grammar.

5.1 From SGML document to AG

The abstract grammar for the letter SGML document is shown below (together with some comments):

```
root letter;
letter : Letter(type idiom date newSymb message end);
```

- the first line indicates that the non-terminal "letter"is the axiom of the grammar (the root of the document's structure).
- the second line represents the production (derivation rule) for the non-terminal "letter"; notice the "Letter"operator in the right hand side - SGen will use it to address the contents of this production during processing; notice also the symbols "type"and "idiom", in the DTD they are defined as attributes of the element "letter- in the translation to a grammar, both attributes and elements become symbols, terminal (T) or non-terminal (N); since attributes are associated with the opening tag in the document instance, in the grammar they will appear in front of the other symbols corresponding the content group.
- the symbol "newSymb"has replaced the content "Name+"(see below).

```
date : DateNull()
     | Date(text);
```

- the element "Date"in the DTD is "PCDATA"; in the grammar it becomes a terminal symbol with its derivation associated with a "text"lexeme.

- in a normal grammar only one production would be used but here we are talking about incremental parsing; even with incremental parsing possibilities the parser needs to create a complete instance of the structure, in this case this is done using "null"productions ("date : DateNull()"); "DateNull()"will be associated with a representation of "date"that will show as "date"content while this element is not filled by the user.

```
list newSymb;
newSymb : NewSymbNull()
        | NewSymb(name newSymb);

name : NameNull()
     | Name(text);
```

- the "Name"element appears in the DTD with an occurrence indicator "+", meaning that this element can be instanciated one or more times.
- elements with cycling occurrence indicators are translated into a recursive production "newSymb : NewSymb(name newSymb)".
- the element "Name "without the occurrence indicator is a normal terminal symbol with its respective productions: the "null"production and the "text"production.
- so, a symbol with a cycling occurrence idicator is translated into two sets of productions; one is recursive and deals with the repeating factor; the other is the normal derivation of the element being affected by the occurrence indicator.

```
message : MessageNull()
        | Message(text);

end : EndNull()
      | End(text);
```

- the elements "Message"and "End"are "PCDATA; thus they are treated as terminal symbols.

```
type : TypeNull()
     | TypeFriend()
     | TypeFamily()
     | TypeWork();

idiom : IdiomNull()
      | IdiomEng()
      | IdiomFre()
      | IdiomSpa()
      | IdiomPor();
```

- the symbols "type"and "idiom"are terminals and correspond the attributes with the same name.
- attributes are always mapped into terminal symbols since their value is never structured.
- in this case we have attributes with enumerated values so we have a production for each possible value besides the "null"production.

5.2 From DTD to AG

Thus far, we have seen the correspondance between a SGML document and an attribute grammar. Now, we shall generalize and in the process build a translation between DTDs and attribute grammars.

Most of the conversion rules were already applied in the previous example. We will look at them again in a generalized perspective.

If we take the "Letter"DTD and generate an attribute grammar, we will end up with the grammar displayed in figure 6.

It is easy to notice that there are differences between this grammar and the one generated earlier. Most of the differences are in the symbol identifiers and this brings us to the first problem we had to solve: the overlap of identifiers.

SGML syntax lets attributes, elements and entities have the same generic identifiers. Even more, attributes associated with different elements can have the same generic identifier. In the new attribute grammar they will all belong to the same domain (terminal and non-terminal symbols) and we must be able to distinguish between them. We achieve this converting the generic identifiers following a set of rules:

- The identifiers of the new symbols and respective production operators will include the original identifier (of the element or attribute) with some prefix and suffix.
- For elements "n"is used as the prefix for non-terminal and terminal symbols and "N"as the prefix for the production operator. In the case of element "Letter", "nLetter"will be the identifier of the non-terminal symbol and "NLetter"will be the identifier of the production operator (see figure 6).

  For attributes "a"is used as the prefix for the symbol and "A"as the prefix for the operator. The "Type"attribute would be converted into "aType"and respective operator would be "AType". But this is not enough since every element can have an attribute with the same identifier. So, to distinguish between them we add a numeric suffix (an integer), that is incremented every time it is used during the generation process. "Type"is converted into "aType0"with its operator being "AType0".
- For entities we use the prefixes "e"and "E". So the entity "sign"would be "esign"with operator "Esign".

Let's now take a look at an element derivation process.

5.2.1 Productions corresponding to elements:

Let `elId` be the identifier of an element; let `attList` be the list of non-terminal symbols corresponding to its attributes; let `elem_content` represent the content of the element `elId`. The productions of a non-terminal symbol corresponding to an element depend on its content. In case it is an `empty` content, a `#PCDATA` content or structured content, the productions are, respectively:

```
n_elId: N_elId_Null()
      | N_elId(attList);
n_elId: N_elId_Null()
      | N_elId(attList TEXT);
n_elId: N_elId_Null()
      | N_elId(attList elem_content);
```

In the first case, the content is empty, so the production's right hand side will only have the list of attributes. In the second case, there is the same list and also `TEXT`, as the content is `#PCDATA`. In the last case, on the right side there is the attribute list `attList` followed by the symbol `elem_content`. As an example consider the first production of our case study (Letter):

```
dtd.syn:*untitled*[NovaTad]
File  Edit  View  Tools  Options  Structure  Text  elp

/*Gramatica Abstracta do Novo Editor*/

nLetter: NLetterNull()
|NLetter(aType0 aIdiom4 nDate oon1 nMessage nEnd );

list oon1;
oon1: Oon1Null()
| Oon1(nName oon1);

aType0: AType0Null()
| AWork1()
| AFamily2()
| AFriend3();

aIdiom4: AIdiom4Null()
| AEnglish5()
| AFrench6()
| ASpanish7()
| APortug8();

nDate: NDateNull()
| NDate(text);

nName: NNameNull()
| NName(text);

nMessage: NMessageNull()
| NMessage(text);

nEnd: NEndNull()
| NEnd(text);

ent : EntNull()
| Ent0(esign);

text : TextNull()
|Text(TEXT)
|TextEnt(ent);

esign: EsignNull()
|Esign: Esign(esign);

Context: dtd
```


Figura 6: **Generated Abstract Grammar**

```
letter : Letter(type idiom date newSymb message end);
```

that corresponds to the following DTD element definition:

```
<!ELEMENT  Letter  - - (Date, Name+, Message, End)>
```

Its attributes are declared as:

```
<!ATTLIST Type (Work | Family | Friend) Work>
<!ATTLIST Idiom (English | French | Spanish| Portug) English>
```

The productions where the non-terminal symbol referent to Letter is derived are:

```
nLetter: NLetter(nType nIdiom nDate newSymb nMessage nEnd);
```

In this production the first and second symbols on the right correspond to the attributes (the `attList` mentioned before). The last sequence of non-terminal symbols is related to the content of Letter, substituting the `elem_content` symbol mentioned before.

To derive the content of an element, first it is necessary to analyse the occurrence indicator (`zerone` for zero or one occurrences, `zeron` for zero or more occurrences and `onen` for one or more occurrences) and then the connector (`alternative` or `sequential`). After this, we repeat the analysis for each operand of the connector, and so on until the last atomic operands (without connectors and occurrence indicators).

Concerning the occurrence indicator, two situations might happen:

- if it is not there, nothing is done and we start to analyse the connector of the same content.
- if it is present, a new non-terminal symbol is created and called `oze`, `ozn` or `oon`, accordingly to the zerone, zeron or onen.

The identifiers of these symbols start with an `o` that is the first letter of occurrence, followed by the first and last letter of the corresponding occurrence indicator.

As it is possible that several new symbols may appear, to avoid repetitions we use an integer suffix `-ne-` that is incremented each time it is used. Let `occur_content` represent the non-terminal symbol corresponding to the argument of an occurrence indicator.

For the `zerone` we have the following productions:

```
oze_ne: Oze_ne_Null()
      | Oze_ne_Nil()
      | Ozer_ne(occur_content);
```

The second production enables the content to occur zero times, and the third one time.

The `zeron` has the following derivation:

```
ozn_ne: Ozn_ne_Null()
      | Ozn_ne_Nil()
      | Ozn_ne(ozn_ne_l);
```

The second production represents the possibility of zero occurrences of the content. In the third production, the symbol `ozn_ne_l` represents one or more occurrences of content. Thus, in SSL (Synthesizer Specification Language) syntax, `ozn_ne_l` is defined as a list. Its productions are:

```
list ozn_ne_l;
ozn_ne_l: Ozn_ne_Null()
        | Ozn_ne_l(ozn_ne_ll ozn_ne_l);
```

The last production has on the right hand side two symbols: the first one (`ozn_ne_ll`) stands for one occurrence of content; the second one (`ozn_ne_l`) is a recursive call.

Finally, the content is derived by the symbol `ozn_ne_ll`:

```
ozn_ne_ll: Ozn_ne_llNull()
         | Ozn_ne_ll(occur_content);
```

And, `onen` derives in the following way:

```
list oon_ne;
oon_ne: Oon_ne_Null()
      | Oon_ne(oon_ne_l oon_ne);

oon_ne_l: Oon_ne_lNull()
        | Oon_ne_l(occur_content);
```

The symbol `oon_ne` is a list of `oon_ne_l`, which represents the content that is repeated one or more times.

In our example, the content of the element Letter has an `oon` occurrence indicator in the element `Name`. So, it will be created a new symbol called `oon1` instead of `newSymb`. Assuming that the identifier of the non-terminal symbol referent to `Name` is `nName`, the productions where the new symbol is derived are:

```
list oon;
oon1: Oon1Null()
    | Oon1(nName oon1);
```

Concerning the connector, if it is sequential, the non-terminal symbols that represent each operand of the connector are sequenced in the production where the element is derived, in the same order they appear in its content. This is the case of the production where `nLetter` is derived, as we showed before.

If the connector is alternative, the new non-terminal symbol (before called `newSymbol`) is called `or` and derived in a set of productions, one for each operand. Again, we use the suffix `ne` to avoid repetitions between symbols of this type. Let `or_ne` be the identifier of this new symbol and `op_n` be the operand that appears in the nth position in the sequence of operands. The productions are:

```
or_ne: Or_ne_Null()
     | Or_ne(op_1)
     | Or_ne+n-1(op_n);
```

5.2.2 Productions corresponding to attributes

When an attribute is enumerated, it has a set of possible values. Let `attId` be the identifier of an SGML attribute. The correspondent non-terminal symbol will have the prefix `a` and the sufix `na` resulting in `a_attId_na`. For a set of `n` possible values for the attribute `attId`, the following productions where `a_attId_na` is derived correspond each one to a value:

```
a_attId_na: A_attId_na_Null()
          | A_attId_na()
          | A_attId_na+n-1();
```

Constant values are represented by empty productions. So, attributes with a possible set of constant values (enumerated) are represented as a set of empty productions. In our earlier example the attribute `Type` is represented by:

```
AType: ATypeNull()
     | ATypeWork()
     | ATypeFamily()
     | AtypeFriend();
```

If the attribute is numerical its productions are:

```
a_attId_na: A_attId_na_Null()
          | A_attId_na(INT);
```

Here, `INT` is the lexeme that represents a integer.

If the attribute is CDATA its productions are:

```
a_attId_na: A_attId_na_Null()
          | A_attId_na(STR);
```

Here, `STR` is the lexeme that represents a string.

`#IMPLIED` attributes are optional. So, they will have, not only the productions corresponding to their values, but also an empty production. Assuming the identifier of this production operator having a suffix `Nil`, the productions are:

```
a_attId_na: A_attId_na_Null()
          | A_attId_na_Nil()
          | A_attId_na()
          | A_attId_na+n-1();
```

5.2.3 Productions corresponding to entities

When entities are declared in the DTD, the user can use them in the text when editing the correspondent instances. So, the symbol that represents any text `-text-` must be derived in `TEXT`, the lexeme that represents any text, and alternatively in the symbol that represents all the declared entities. This last symbol is called `ent` and is derived in several productions, each one representing an entity. The non-terminal symbol corresponding to an entity is derived in the lexeme that represents the way an entity is referenced in the SGML document, that is between `&` or `%` and `;`.

Let `e_entId1` and `e_entIdn` be the identifiers of the non-terminal symbols referent to the entities `entId1` and `entIdn`. Finally we have the following productions:

```
Text : TextNull()
     | Text(TEXT)
     | Text(ent);

ent : EntNull()
    | Ent_entId1(e_entId1)
    ...
    | Ent_entIdn(e_entIdn);
```

```
e_entId1: E_entId1Null()
        | E_entId1();

        ...

e_entIdn: E_entIdnNull()
        | E_entIdn();
```

From the DTD Letter we can extract an example based on the entity `sign`. The corresponding productions are:

```
Text : TextNull()
     | Text(TEXT)
     | Text(ent);

ent : EntNull()
    | Entsign(esign);

esign: EsignNull ()
     | Esign();
```

5.3 SGML Specials

There are some SGML features that we did not cover in our system. That was due to the complexity in the implementation of the solutions that we have considered. We shall discuss some of those cases and respective solutions.

5.3.1 And connector (&)

The operands of a `&` connector appear in the content in any order, but both have to appear. So, it is equivalento to a model group with a `or (|)` connector which operands are the two possible sequences of the operands.

For example,

```
<!ELEMENT  Date  - -  ((Day & Month) , Year)>
```

is equivalento to:

```
<!ELEMENT Date  - -  (((Day, Month) | (Month, Day)) , Year)>
```

In the context of AG, since there are no direct implementation for the `&` connector, we have to use the combinatory sequences of operands shown above.

```
nDate :  NDateNull()
      |  NDate1( nDay, nMonth, nYear )
      |  NDate2( nMonth, nDay, nYear );
```

5.3.2 Inclusions

It may happen that an element can occur in all the sub-elements of a given element. To include that element in all the model groups would be cumbersome. The same effect may be achieved via an inclusion of elements that are not logically part of the document's hierarchy structure [Her94]. It can be said that inclusions can also be an

accident waiting to happen because of the far-reaching implications they have for the overall structure of the DTD [McGrath98].

Basically, every model group occurring below the inclusion point in the hierarchy inherits the inclusion. So, at the time of creating the new AG, it would be necessary to extend the productions of every non-terminal symbol that represents an element in the model content. This way, we allow the included element to appear anywhere in the content.

For example, the following declarations mean that a footnote may appear anywhere in the content of the letter element.

```
<!ELEMENT  Letter  - -  (Date, Name+, Message, End)  +(Fn)>
<!ELEMENT  Date  - - (#PCDATA)>
...
```

An equivalent way of expressing the same but without inclusions would be:

```
<!ELEMENT  Letter  - -  (Fn*,Date,Fn*,Name+,Fn*,Message,Fn*,End,Fn*)>
<!ELEMENT  Date  - -  (#PCDATA|Fn)*>
...
```

The equivalent AG would be:

```
nLetter: NLetterNull()
       | NLetter( nFnSeq, nDate, nFnSeq, nNameSeq, nFnSeq,
                  nMessage, nFnSeq, nEnd, nFnSeq );

list nDate;
nDate: NDateNull()
     | NDate( newsymb, nDate );

newsymb: NewSymbNull()
       | NewSymbNil()
       | NewSymb1( nFn )
       | NewSymb2( TEXT );
```

The symbol `nFnSeq` represents the footnote element ocurring zero or more times.

Analogous productions to the ones associated with `nDate` should be generated for the remaining non-terminal symbols in the letter content.

5.3.3 Exclusions

Exclusions are analogous to inclusions but have the effect of excluding specified elements from model contents. Perhaps their most common application is the removal of unwanted recursion in model contents [McGrath98].

In the previous example we can define a footnote as a message and use an exclusion to avoid a footnote inside itself.

```
<!ELEMENT  Message  - -  (#PCDATA | Fn)*>
<!ELEMENT  Fn  - - (Message)  -(Fn)>
```

Concerning the AG being generated, we could give a solution to this problem but this would imply a complex explosion in the number of productions that would have to be added.

One other solution is to leave the grammar untouched and provide semantic constraints to achieve the same goal. In this case we must check if the message (the parent) where the footnote appears is inside a footnote, if so, an error message would be issued. This context rule would be associated with `nMessage` productions.

5.3.4 Tag Omission

Structured editors are driven by an abstract syntax grammar. In terms of SGML, tags are only present to signal element boudaries, they are syntactic sugar in the abstract syntax grammar. So, in our editors tag omission represents a decision about the visual interface. There is no need for them to be present. The minimization rules in element declaration work as switches to the visual appearance.

6 Conclusion

We have presented the architecture and the underlying principles of a Document Programming Environment we are developing (fig. 2). Following a typical compilers development approach we are using attribute grammar based specifications to implement the various pieces of the system.

We have discussed in more detail the central component of the system, a DTD editor that automatically generates attribute grammars corresponding to the DTDs being edited.

With this work we introduced concepts like incremental parsing and on-line validation. These are improvements to the traditional SGML editor/parser environments.

This metodology opened a gate to semantics processing. Attribute Grammars give a possible formal approach to semantics.

We are extending this system to process semantics, embedding a constraint language into the SGML syntax and plugging in INES a new module to verify the constraints.</PARA>

7 Acknowledgements

Thanks are dued to JNICT and PRODEP for the grant under which this work is being developed, and to Filomena Louro for proof reading this paper.

Referências

[Her94] Eric van Herwijnen, *"Practical SGML"*, Kluwer Academic Publishers, 1994.

[Knu68a] Donald E. Knuth, *"Semantics of context-free languages"*, Mathematical Systems Theory, 2(2):127–145, 1968.

[McGrath98] Sean McGrath, *"Parseme.1st"*, Prentice Hall, 1998.

[RH98] Ramalho, J. C., and Henriques, P. R., *"Beyond DTDs: constraining data content"*, In proceedings of "SGML/XML Europe 98"conference, Paris, May 1998.

[RT89a] Thomas Reps and Tim Teitelbaum, *"The Synthesizer Generator: A System for Constructing Language-Based Editors"*, Texts and Monographs in Computer Science, Springer-Verlag, 1989.

[RT89b] Thomas Reps and Tim Teitelbaum, *"The Synthesizer Generator Reference Manual"*, Texts and Monographs in Computer Science, Springer-Verlag, 1989.